Corrector:	DANIEL		
Nota Final / Ejs:	1	2	3
Aprobado	P	P	A

Algoritmos y Estructuras de Datos II
Segundo parcial – 10 de Junio de 2022

Aclaraciones

- El parcial es a libro casi-cerrado. Solo es posible tener impreso el teorema maestro y el apunte de diseño. Además, pueden tener una hoja (2 carillas), escrita a mano, con los apuntes que se deseen.
- Cada ejercicio debe entregarse en **hojas separadas**. Las mismas deben estar numeradas.
- Incluir en esta hoja: nombre, apellido, el número de orden asignado, número de libreta.
- Cada ejercicio se calificará con Perfecto, Aprobado, Regular, o Insuficiente.
- El parcial estará aprobado si las notas de los tres ejercicios tienen al menos dos A.
- Los ejercicios no se recuperan por separado.
- Se encuentran disponibles para utilizar todas las estructuras de datos presentadas en la teórica con las operaciones y complejidades dadas en las mismas.

Ej. 1.

Proponer un algoritmo que permita ordenar un arreglo de naturales (sobre los que no se conoce nada en especial) de manera ascendente en $O(n \log d)$ en el peor caso, donde donde n es la cantidad de elementos a ordenar y d es la cantidad de elementos distintos. Justifique informalmente por qué su algoritmo cumple con lo pedido.

Ej. 2.

Se tiene una secuencia de alturas $h_1, \ldots, h_n$. Decimos que un intervalo $h_t, h_{t+1}, \ldots, h_{t+k}$ es un *edificio* si para todo $i \in [t, t+k)$, el valor $|h_i - h_{i+1}|$ no es mayor que una cierta tolerancia dada θ y además tanto $|h_t - h_{t-1}|$ como $|h_{t+k} - h_{t+k+1}|$ (en caso de existir) son mayores que θ. El *ancho* de un edificio es la cantidad de alturas que lo componen. Por ejemplo, si la secuencia de alturas es

$$[100, 101, 100, 103, 80, 77, 74, 200, 32, 30]$$

y $\theta = 3$, entonces los edificios de esta secuencia de alturas serían $[100, 101, 100, 103]$, $[80, 77, 74]$, $[200]$, $[32, 30]$, y sus anchos serían 4, 3, 1 y 2, respectivamente.

Escribir un algoritmo que tome un arreglo de enteros (que representan alturas) y una tolerancia θ y devuelva las mismas ordenadas. El orden estará dado según los anchos de los edificios en forma creciente. Tanto los edificios como las alturas dentro de cada edificio deben mantener el orden original. En el ejemplo anterior, el resultado esperado sería $[200, 32, 30, 80, 77, 74, 100, 101, 100, 103]$. La complejidad del algoritmo no debe ser peor que $O(n)$, donde n es la cantidad de alturas dada. Justifique informalmente la correctitud del algoritmo y su complejidad temporal.

Ej. 3.

Se tiene un arreglo ordenado de n números naturales consecutivos con k *huecos*, es decir, en el arreglo están presentes todos los elementos dentro de un rango determinado salvo una cantidad k de ellos. Por ejemplo, el arreglo $A = [11, 12, 13, 15, 16, 19, 20, 21]$ tiene *huecos* en los valores $[14, 17, 18]$. Se pide describir un algoritmo que devuelva una lista con todos los *huecos* de un arreglo. Se puede asumir que el arreglo tiene tamaño potencia de 2.

a) Dar un algoritmo que use la técnica de *Divide and Conquer* y resuelva el problema en tiempo $O(k \log(n))$ en el peor caso.
b) Marcar claramente qué partes del algoritmo se corresponden a *dividir*, *conquistar* y *unir* subproblemas.
c) Asumiendo que $k = 1$, justificar formalmente que el algoritmo cumple con la complejidad pedida.
d) Justificar informalmente, para el caso general $k > 1$, que la complejidad es $O(k \log(n))$.

```
Ordenar (in/out A: arreglo(nat))
  D: diccAVL(nat, nat) ← Vacio()                          O(1)
  For i:nat ← 1 to tam(A)                                 O(n log d)
    if Definido?(D, A[i]) then                            O(log d)
      Definir(D, A[i], Significado(D, A[i]) + 1)          O(2 log d)
    else
      Definir(D, A[i], 1)                                 O(log d)
    end if
  end for
  C: lista(nat) ← Claves(D)                               O(d)
  it ← CrearIt(C)                                         O(1)
  K:nat ← 1                                               O(1)
  while HaySiguiente(it)                                  O(n log d) *
    clave: nat ← Siguiente(it)                            O(1)
    reps: nat ← Significado(D, clave)                     O(log d)
    for i:nat ← 1 to reps                                 O(reps)
      A[K] ← clave                                        O(1)
      K ← K + 1                                           O(1)
    end for
    Avanzar(it)                                           O(1)
  end while
```

* ABUSO DE NOTACIÓN. HAY d CLAVES Y POR CADA UNA HAY reps ≥ 1 SI SUMAMOS LAS ITERACIONES DEL while Y DEL for INTERNO TENEMOS EN TOTAL n ITERACIONES, LA CANTIDAD ORIGINAL DE ELEMENTOS. EL MAYOR COSTO DE UNA ITERACIÓN ES O(log d).

LA FUNCIÓN Claves DE MI DICCIONARIO AVL DEVUELVE LAS CLAVES EN ORDEN. SI HAY d CLAVES, LA COMPLEJIDAD ES O(d) PUES HAY QUE VISITAR CADA NODO DEL AVL UNA VEZ, UTILIZANDO EL ALGORITMO InOrder.

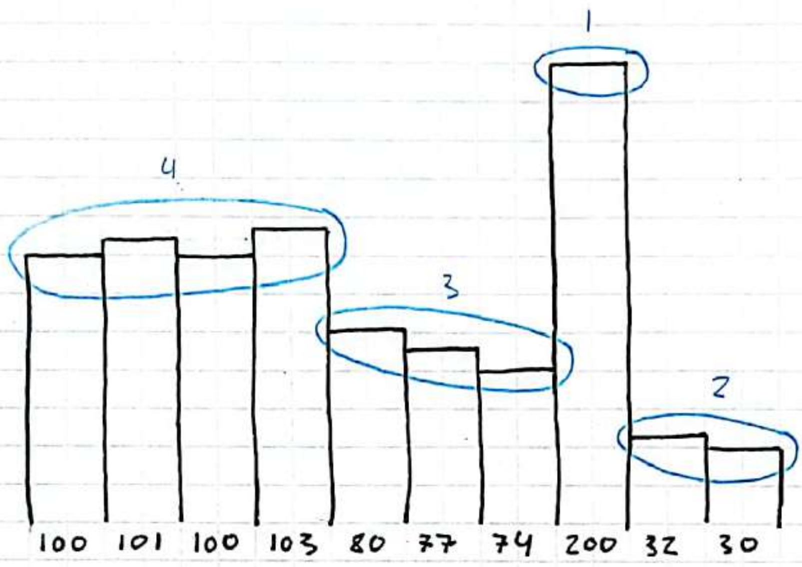
```
Claves (in D: diccAVL(nat, nat)) → res: lista(nat)
  res ← Vacía()                                  O(1)
  InOrder(D.raiz, res)                           O(d)

InOrder (in N: nodoAVL, in/out res: lista(nat))
  if N ≠ nil then                                O(1)
    InOrder(N.izq, res)                          O(tam(N.izq))
    AgregarAtras(res, N.clave)                   O(1)
    InOrder(N.der, res)                          O(tam(N.der))
  end if
```

LA COMPLEJIDAD DE LA FUNCIÓN Ordenar RESULTA O(n log d). EL ALGORITMO PRIMERO RECORRE EL ARREGLO DE ENTRADA DE n ELEMENTOS, Y POR CADA UNO, DEFINE O INCREMENTA LA CANTIDAD DE REPETICIONES EN EL DICCIONARIO AVL. POR SER UN AVL, LAS OPERACIONES DE BÚSQUEDA/INSERCIÓN SON O(d) DONDE d ES LA CANTIDAD DE NODOS DEL AVL, ES DECIR, LA CANTIDAD DE ELEMENTOS DISTINTOS DEL ARREGLO DE ENTRADA (NO HAY CLAVES REPETIDAS EN EL AVL).

LUEGO, UTILIZANDO EL ALGORITMO InOrder SE RECUPERAN TODAS LAS CLAVES DEL DICCIONARIO EN O(d). AHORA TENEMOS EL ARREGLO CASI ORDENADO. FALTA RECORRER LAS d CLAVES, Y POR CADA UNA, OBTENER EL SIGNIFICADO, ES DECIR LA CANTIDAD DE REPETICIONES DE ESE ELEMENTO, E INSERTARLO ESA CANTIDAD DE VECES EN EL ARREGLO.

HUBIESE ITERADO EL DICCIONARIO DIRECTAMENTE PERO NO SABÍA CÓMO EXPLICAR BIEN CÓMO OBTENGO LAS CLAVES EN ORDEN. LA COMPLEJIDAD SERÍA LA MISMA PERO CON UNA CONSTANTE MENOR.

muy bien :)



COMO MÁXIMO PUEDEN HABER n EDIFICIOS DISTINTOS. ARMAMOS n BUCKETS DE LISTAS DE ALTURAS (nat). RECORREMOS LA SECUENCIA DE ALTURAS Y VAMOS AGRUPANDO LAS ALTURAS QUE FORMAN UN EDIFICIO. CUANDO DETECTAMOS QUE ARRANCA OTRO EDIFICIO O LLEGAMOS AL FINAL, VEMOS CUÁNTAS ALTURAS FORMAN ESE EDIFICIO. ESA CANTIDAD ES EL ANCHO DEL EDIFICIO. AGREGAMOS ESTA SUBSECUENCIA DE ALTURAS AL BUCKET EN LA POSICIÓN DEL ANCHO DEL EDIFICIO.

PARA ARMAR EL RESULTADO, RECORREMOS LOS BUCKETS DE FORMA CRECIENTE Y CONCATENAMOS LAS ALTURAS PARA CADA ANCHO.

AL AGREGAR UN NUEVO EDIFICIO A SU BUCKET, LO AGREGAMOS AL FINAL PARA MANTENER EL ORDEN RELATIVO ENTRE EDIFICIOS DEL MISMO ANCHO.

```
OrdenarEdificios (in/out A: arreglo(nat), d: nat)
  n ← tam(A)                                                    O(1)
  buckets: arreglo(lista(nat)) ← CrearArreglo(n)                O(n)
  For i ← 1 to n                                                O(n)
    buckets[i] ← Vacía()                                        O(1)
  end For
  t ← 1                                                         O(1)
  while t ≤ n                                                   O(n)
    K ← t                                                       O(1)
    E: lista(nat) ← Vacía()                                     O(1)
    AgregarAtras(E, A[K])                                       O(1)
    while K < n ∧L |A[K] - A[K+1]| ≤ d                          O(n-t)
      K ← K+1                                                   O(1)
      AgregarAtras(E, A[K])                                     O(1)
    end while
    t ← K+1                                                     O(1)
    ancho ← Longitud(E)                                         O(1)
    ConcatenarAtras(buckets[ancho], E)                          O(1)
  end while
  K ← 1                                                         O(1)
  For ancho ← 1 to n                                            O(n)
    it ← CrearIt(buckets[ancho])                                O(1)
    while HaySiguiente(it)                O(Longitud(buckets[ancho]))
      A[K] ← Siguiente(it)                                      O(1)
      Avanzar(it)                                               O(1)
      K ← K+1                                                   O(1)
    end while
  end For
```

AED 2 4
EJERCICIO 2 10/6/22

LA COMPLEJIDAD RESULTA O(n).

CREAR n BUCKETS CON LISTAS VACÍAS ES O(n)

RECORRER LAS n ALTURAS BUSCANDO LOS EDIFICIOS ES O(n).

SI BIEN HAY UN CICLO INTERNO, LA CANTIDAD DE ELEMENTOS QUE ÉSTE CICLO RECORRE LUEGO LOS SALTEAMOS EN EL CICLO EXTERIOR. EN EFECTO SE VISITA UNA SOLA VEZ CADA ELEMENTO.

FINALMENTE, REUBICAR LOS ELEMENTOS EN SUS LUGARES ORDENADOS ES O(n) PORQUE TAMBIÉN SE VISITA CADA ELEMENTO UNA SOLA VEZ, Y HAY n ELEMENTOS EN TOTAL.

AL RECORRER LOS n BUCKETS, CIERTA CANTIDAD DE ÉSTOS CONTIENEN LISTAS VACÍAS, Y POR LO TANTO NO HACEMOS NINGUNA ITERACIÓN EN EL CICLO INTERNO CON EL ITERADOR.

PARA LOS BUCKETS QUE SÍ TIENEN ELEMENTOS, EL COSTO DE COLOCAR ESOS ELEMENTOS EN SUS LUGARES ES O(K) DONDE K = CANTIDAD DE ELEMENTOS EN EL BUCKET.

DICHO DE OTRA FORMA:

$$\sum_{i=1}^{n}\left(\sum_{K=1}^{\text{Longitud(buckets[i])}} \underbrace{1}_{\text{COSTO DE HACER ALGO CON CADA ELEMENTO}}\right) = n$$

muy bien :)

AED2
EJERCICIO 3

5
10/6/22

```
Huecos (in A: arreglo(nat)) → H: lista(nat)
  if tam(A) ≤ 1 then
    H ← Vacía()
  else
    H ← HuecosAux(A, 1, tam(A))
  end if

HuecosAux (in A: arreglo(nat), in low: nat, in high: nat) → H: lista(nat)
  H ← Vacía()
  if A[high] - A[low] ≠ high-low then    // Hay huecos
    if low+1 = high then    // Caso base: buscar huecos entre 2 elementos
CONQUISTAR {
      for K ← A[low]+1 to A[high]-1
        AgregarAtras(H, K)
      end for
    }
    else    // Recursión
DIVIDIR {
      Hizq ← HuecosAux(A, low, (low+high)/2)
      Hder ← HuecosAux(A, (low+high)/2 + 1, high)
    }
UNIR { H ← Concatenar(Hizq, Hder)
    end if
  end if
```

bien

Acá me olvidé de buscar los posibles huecos entre:
A[(low + high) / 2] y A[(low + high) / 2 + 1]
Habría que hacer el mismo for que está en la parte de conquistar (lo pondría como una función aux para usarla allá y acá).

EL COSTO DE HUECOS ES:

$T(n) = 2T(\frac{n}{2}) + K$ No exactamente. Tenés casos base de tamaño arbitrariamente grandes

DONDE K ES LA CANTIDAD DE HUECOS.

SUPONGAMOS QUE $K=1$. DEBIDO AL PRIMER if EN HuecosAux, SOLO VAMOS A RESOLVER UNO DE LOS DOS SUBPROBLEMAS, YA QUE SI HAY UN ÚNICO HUECO, VA A ESTAR EN LA MITAD IZQUIERDA DEL ARREGLO O EN LA MITAD DERECHA.

POR LO TANTO, SI $K=1 \Rightarrow T(n) = T(\frac{n}{2})$ claro

SEAN $a=1$, $b=2$, $f(n)=0=\Theta(1)$

QVQ $f(n) = \Theta(n^{\log_b(a)})$

$\Theta(n^{\log_b(a)}) = \Theta(n^{\log_2(1)}) = \Theta(n^0) = \Theta(1)$

POR TEO. MAESTRO, $T(n) = \Theta(n^{\log_2(1)} \log n) = \Theta(\log n)$

OBSERVEMOS QUE SI $K=1 \Rightarrow \Theta(K \log n) = \Theta(\log n)$

bien

INFORMALMENTE, SI $K>1$ EL COSTO DE CONQUISTAR SERÁ $\Theta(K)$ PUES HAY QUE AGREGAR LOS K HUECOS A LA LISTA. UNIR ES $O(1)$ YA QUE UNIR LISTAS ENLAZADAS ES $O(1)$ SI NO NOS INTERESA MANTENER LAS LISTAS ORIGINALES.

SI EN CADA PASO RECURSIVO DIVIDIMOS EL PROBLEMA POR LA MITAD, VEAMOS QUE EL TAMAÑO DEL PROBLEMA DECRECE DE ESTA FORMA HASTA LLEGAR A $n=2$:

$\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \frac{n}{16}, \dots, \frac{n}{2^i}$ DONDE i ES LA I-ÉSIMA ITERACIÓN.

QVQ $\frac{n}{2^i} = 2 \iff n = 2^{i+1} \iff i+1 = \log(n)$

$\Rightarrow$ HACEMOS $\log(n)-1$ PASOS RECURSIVOS Y EL COSTO DEL CASO BASE ES K. POR LO TANTO, PARA $K>1 \Rightarrow O(K \log n)$

El truco es que no siempre. Si fuera así, la complejidad resultante sería $O(nK)$.

Queda $O(K \log n)$ porque sólo tenés $O(K)$ casos base.